DEPSTAR

Devang Patel Institute of Advance Technology and Research

(A Constitute Institute of CHARUSAT)

Certificate

This is to certify that

Mr./ ~~Mrs.~~ Probin Bhagchandani

of 6CE1 *Class,*

ID. No. 22DCE006 *has satisfactorily completed his/ her term work in* OCCE3001 - Deep Learning *for the ending in* April *2024 /2025*

K.S. Patel

Dgang

Date :

Sign. of Faculty

Head of Department

CHARUSAT
CHAROTAR UNIVERSITY OF SCIENCE AND TECHNOLOGY

Faculty of Technology and Engineering

B.Tech (CE/IT/CSE)

Date: 16/12/2024

Practical List

Academic Year	:	2024-2025	Semester	:	6
Course code	:	OCCE3001	Course name	:	Deep Learning

Sr. No.	Aim
1.	**Solve the XOR Problem Using Multilayer Perceptrons (MLPs)** You are required to demonstrate how Multilayer Perceptrons (MLPs) solve non-linear problems like the XOR problem, showcasing their ability to handle complex decision boundaries. Follow these steps: 1. Design a Neural Network: Create a two-layer neural network with one hidden layer. Use a non-linear activation function (e.g., ReLU, sigmoid, or tanh) in the hidden layer. 2. Understand the Challenge: Begin by exploring why single-layer perceptrons cannot solve the XOR problem. Compare the limitations of single-layer perceptrons with the capabilities of MLPs. 3. Train the Network: Implement gradient descent to optimize the network's weights. Use appropriate loss and optimization functions to train the network on the XOR dataset. 4. Visualize the Decision Boundary: Once the network is trained, plot and interpret the decision boundary to understand how the MLP classifies the XOR problem. This task will help you understand the importance of hidden layers and non-linear activation functions in solving non-linear problems.
2.	**Implement and Compare Gradient Descent Variants** In this task, you will explore and compare the performance of different gradient descent optimizers. The objective is to understand how these variants affect training speed, convergence, and final accuracy. You will work with the MNIST Handwritten Digits dataset for this experiment. Follow these steps: 1. Choose Optimization Techniques: Implement the following gradient descent optimizers: a. Stochastic Gradient Descent (SGD) b. Momentum Gradient Descent (Momentum GD) c. RMSProp d. Adam

	2. Train the Model: Use each optimizer to train a neural network on the MNIST dataset. Ensure you keep all other factors, such as the network architecture and initial learning rate, consistent for a fair comparison. 3. Analyze Performance: a. Compare the training speed of each optimizer by observing how quickly they reduce the loss. b. Analyze the convergence behavior to see which optimizer reaches the minimum loss most effectively. c. Evaluate the final accuracy of the model trained with each optimizer. 4. Hyperparameter Tuning: Experiment with hyperparameters (e.g., learning rates, decay rates) for each optimizer to observe how they influence performance. 5. Summarize Insights: Document the key differences among the optimizers and their suitability for various types of problems. This task will help you understand the significance of choosing an appropriate optimization technique and how it impacts model performance and training efficiency.
3.	**Implement the Backpropagation Algorithm** In this task, you will implement the backpropagation algorithm from scratch and understand how gradients are calculated for weight and bias updates in a neural network. You will work with a Simple Binary Classification Dataset (e.g., churn data converted to a binary problem). Follow these steps: 1. Set Up the Dataset: Prepare the churn dataset, ensuring it is formatted as a binary classification problem. Split the data into training and testing sets. 2. Design the Neural Network: Create a simple Multilayer Perceptron (MLP) with one hidden layer. Use appropriate activation functions for the hidden layer (e.g., ReLU, sigmoid, or tanh) and output layer (e.g., sigmoid for binary classification). 3. Implement Forward Propagation: Write the forward pass to calculate the outputs of the network layer by layer, including the final predicted outputs. 4. Implement Backpropagation: a. Derive the gradients of the loss function with respect to the weights and biases. b. Implement the chain rule to propagate errors backward through the network. c. Update the weights and biases using the calculated gradients and a chosen learning rate. 5. Train the Model: Use the implemented backpropagation algorithm to train the network on the training data. Monitor the loss at each epoch to ensure the model is learning. 6. Evaluate the Model: Test the model on the testing dataset and evaluate its performance using metrics such as accuracy, precision, and recall. 7. Summarize Insights: Reflect on how the errors propagate backward, how gradients are computed, and how these updates improve the model's performance. This experiment will enhance your understanding of loss functions, activation functions, their gradients, and the essential role of backpropagation in training neural networks.

4.	**Explore the Impact of Regularization, Batch Normalization, and Weight Initialization on Model Training**

In this task, you will explore how regularization, batch normalization, and advanced weight initialization techniques affect the training and generalization of deep networks. You will implement a Convolutional Neural Network (CNN) using any library and experiment with these methods on the CIFAR-10 Image Dataset. Steps to Follow

1. Set Up the Dataset:
   a. Load and preprocess the CIFAR-10 dataset.
   b. Normalize the image data and divide it into training, validation, and test sets.
2. Design the CNN:
   a. Build a CNN architecture suitable for the CIFAR-10 dataset.
   b. Ensure the model includes convolutional, pooling, and fully connected layers.
3. Experiment with Regularization:
   a. Apply techniques like L1/L2 regularization (weight decay) and Dropout.
   b. Train the model with and without regularization, and compare the impact on overfitting and generalization.
4. Add Batch Normalization:
   a. Integrate batch normalization layers after the convolutional and fully connected layers.
   b. Train the model with and without batch normalization, and observe its effect on training stability, convergence speed, and accuracy.
5. Experiment with Weight Initialization:
   a. Use different weight initialization strategies, such as Random Initialization, Xavier Initialization, and He Initialization.
   b. Compare their effects on training, focusing on addressing vanishing and exploding gradients.
6. Train and Compare Models:
   a. Train the models with various combinations of regularization, batch normalization, and weight initialization.
   b. Monitor training and validation performance (accuracy and loss) to evaluate generalization.
7. Analyze Results:
   a. Compare the models' performance based on training stability, convergence speed, and accuracy.
   b. Highlight how each technique contributes to improved model performance.
8. Summarize Insights: Reflect on the role of regularization in preventing overfitting. Discuss how batch normalization stabilizes training and accelerates convergence. Explain the importance of proper weight initialization in avoiding gradient-related issues.

This practical will help you understand critical techniques that improve training and generalization in deep networks, preparing you for advanced model design and optimization.

5. **Dimensionality Reduction Using PCA**

In this task, you will apply Principal Component Analysis (PCA) to perform dimensionality reduction and visualize high-dimensional data in 2D space. You will work with the Wine Dataset to understand how PCA extracts principal components and their significance in explaining data variance. Steps to Follow

1. Set Up the Dataset:
    a. Load the Wine dataset and preprocess it by normalizing or standardizing the features to ensure they are on the same scale.
2. Apply PCA:
    a. Compute the principal components of the dataset.
    b. Reduce the dimensionality of the data to 2 components and visualize it in a 2D scatter plot, labeling the data points by their class.
3. Analyze Principal Components:
    a. Examine the explained variance ratio of each principal component to understand how much of the data variance is captured.
4. Reconstruct Data:
    a. Reconstruct the dataset from the reduced dimensions and compare it with the original data.
    b. Quantify the reconstruction error to observe the trade-off between dimensionality reduction and information loss.
5. Visualize Results:
    a. Plot the original data (if feasible) and the reduced data to visually interpret the difference.
    b. Highlight how PCA separates the classes in the Wine dataset based on its variance-capturing property.
6. Summarize Insights: Reflect on how PCA helps in reducing the complexity of high-dimensional data while retaining most of the variance. Discuss the balance between reducing dimensions and preserving important information.

This experiment will give you practical insights into the use of PCA for dimensionality reduction, its effect on data interpretation, and its importance in simplifying high-dimensional datasets.

6. **Denoising Images Using Autoencoders**

In this task, you will implement a denoising autoencoder to clean noisy images, using the MNIST Handwritten Digits dataset. You will explore the encoding, decoding, and reconstruction processes in neural networks and understand how autoencoders relate to PCA. This experiment will also highlight the role of regularization and noise handling in autoencoders. Steps to Follow

1. Set Up the Dataset:
    a. Load and preprocess the MNIST dataset.
    b. Introduce noise to the images by adding random Gaussian or salt-and-pepper noise. Retain the original images as clean versions for comparison.
2. Design the Autoencoder:
    a. Build a neural network architecture with an encoder that compresses the input images into a lower-dimensional latent space and a decoder that reconstructs the images from the latent representation.
    b. Use non-linear activation functions like ReLU or sigmoid for the layers.
3. Train the Autoencoder:

	a. Train the autoencoder with noisy images as input and clean images as output. b. Use a suitable loss function, such as Mean Squared Error (MSE), to measure the difference between the reconstructed and original images. c. Monitor training loss to ensure the model is learning to remove noise effectively. 4. Test the Autoencoder: a. Evaluate the model on a separate test set of noisy images and compare the reconstructed images with their clean counterparts. 5. Analyze Regularization: a. Experiment with adding regularization techniques, such as L1/L2 penalties or Dropout, to the autoencoder. b. Observe their impact on the reconstruction quality and model generalization. 6. Visualize Results: a. Display side-by-side comparisons of noisy, clean, and reconstructed images. b. Plot the latent space representation to understand how the autoencoder encodes the input data. 7. Summarize Insights: Reflect on how denoising autoencoders handle noise effectively and improve image quality. Discuss the differences between autoencoders and PCA, emphasizing the power of neural networks in capturing non-linear features. This practical will enhance your understanding of autoencoders, their applications in denoising.
7.	**Visualizing CNN Filters and Feature Maps** In this task, you will explore Convolutional Neural Networks (CNNs) by visualizing the filters they learn and the feature maps generated during feature extraction. This will help you understand the hierarchical learning process in CNNs. You will work with two datasets and tasks: Part1: CIFAR-10 Dataset: Train a CNN from scratch to visualize filters and feature maps. Part2: Chest X-ray Pneumonia Dataset: Fine-tune a pretrained CNN (e.g., ResNet50, VGG16, or EfficientNet) for medical image classification and interpret the feature extraction process. **Part 1: Visualizing CNN Filters and Feature Maps (CIFAR-10)** 1. Set Up the Dataset: a. Load and preprocess the CIFAR-10 dataset (e.g., normalize images and split into training, validation, and test sets). 2. Train a CNN: a. Build a CNN architecture from scratch or use a simple predefined model. b. Train the CNN on CIFAR-10 and evaluate its performance on the validation set. 3. Visualize Filters: a. Extract and visualize the learned filters from the convolutional layers. b. Understand how these filters capture basic patterns like edges, corners, and textures. 4. Inspect Feature Maps: a. For a given input image, pass it through the network and capture intermediate feature maps. b. Visualize these feature maps to observe how the network extracts hierarchical features at different layers.

5. Analyze Results: Discuss the progression from low-level features (edges and textures) in early layers to high-level features (shapes and objects) in deeper layers.

Part 2: Fine-Tuning Pretrained CNN Models (Chest X-ray Pneumonia Dataset)

1. Set Up the Dataset:
   a. Load the Chest X-ray Pneumonia dataset and preprocess it (resize images, normalize pixel values, and split into training, validation, and test sets).
2. Choose a Pretrained Model:
   a. Select a pretrained CNN model such as ResNet50, VGG16, or EfficientNet.
   b. Fine-tune the model by replacing the final classification layer with a layer suitable for binary classification (pneumonia vs. normal).
3. Train the Model:
   a. Train the model on the Chest X-ray dataset and evaluate its performance using metrics like accuracy, precision, and recall.
4. Visualize Filters and Feature Maps:
   a. Visualize the filters learned in the convolutional layers of the pretrained model.
   b. Pass a sample chest X-ray image through the network and capture intermediate feature maps to observe feature extraction at different stages.
5. Analyze Results: Compare the filters and feature maps of the pretrained model with those of the scratch-trained CIFAR-10 model.
6. Discuss how pretrained models adapt to specific tasks like medical image classification.

This practical will deepen your understanding of CNNs, hierarchical feature learning, and the power of transfer learning in specialized tasks.

8. **Text Pre-Processing and Word Embedding**

In this task, you will prepare textual data for machine learning models by applying text pre-processing techniques and converting text into numerical vectors using word embedding methods. The goal is to understand how pre-processing cleans and standardizes text and how embeddings like GloVe, Word2Vec, and BERT capture semantic relationships in text data. You will use the IMDB Movie Reviews Dataset for a sentiment analysis task. Steps to Follow

1. Set Up the Dataset:
   a. Load the IMDB Movie Reviews dataset and split it into training and test sets.
2. Perform Text Pre-Processing:
   a. Apply the following steps to clean and prepare the text:
      i. Convert text to lowercase.
      ii. Remove punctuation, numbers, and special characters.
      iii. Tokenize the text into words.
      iv. Remove stopwords using libraries like NLTK or Spacy.
      v. Apply stemming or lemmatization to normalize words.
3. Implement Word Embedding Techniques:
4. Static Embeddings (GloVe and Word2Vec):
   a. Use pre-trained GloVe or Word2Vec models from Gensim to convert each word into a fixed-length dense vector.
   b. Represent each review as an average or sum of its word vectors.

4. Dynamic Embeddings (BERT):
    a. Use the Hugging Face Transformers library to extract contextual embeddings from BERT.
    b. Tokenize the text using BERT's tokenizer and represent each review by the embedding of the [CLS] token or the average of token embeddings.
5. Train a Sentiment Analysis Model:
    a. Use the generated embeddings as input features for a classifier (e.g., logistic regression, random forest, or a simple neural network).
    b. Train separate models for GloVe, Word2Vec, and BERT embeddings to compare their performance.
6. Evaluate Model Performance:
    a. Evaluate the models on the test set using accuracy, precision, recall, and F1-score.
    b. Analyze the impact of different embeddings on the classifier's performance.
7. Analyze and Compare Embeddings:
    a. Highlight the key differences between static (GloVe and Word2Vec) and dynamic embeddings (BERT).
    b. Discuss how BERT considers the context of words, while GloVe and Word2Vec produce the same vector for a word regardless of its context.
8. Summarize Insights: Reflect on the importance of text pre-processing in cleaning and standardizing textual data. Discuss how word embeddings enhance feature representation by capturing semantic meanings. Highlight the strengths of dynamic embeddings like BERT in handling context compared to static embeddings like GloVe and Word2Vec.

This practical will deepen your understanding of text processing, word embeddings, and their application in Natural Language Processing (NLP) tasks.

9. **Implementation of Sequential Models**

In this task, you will explore sequential data processing by implementing Recurrent Neural Networks (RNNs), Gated Recurrent Units (GRUs), and Long Short-Term Memory (LSTM) networks. These models are foundational for capturing temporal dependencies in sequential data. You will address challenges like vanishing gradients in RNNs and see how GRUs and LSTMs overcome them using gating mechanisms. The task involves training and evaluating these models on two datasets for different sequential tasks: Steps to Follow

1. Load and Preprocess the Dataset:
    a. Load the Airline Passenger dataset (e.g., monthly passenger counts).
    b. Normalize the data for efficient training.
    c. Prepare the data for sequence modeling by creating sliding windows of input-output pairs.
2. Implement Sequential Models:
    a. Build and train the following sequential models:
        - RNN: A basic recurrent neural network to learn temporal dependencies.
        - GRU: A gated model addressing RNN limitations by selectively retaining information.
        - LSTM: A model with forget and input gates to capture long-term dependencies effectively.
3. Train the Models:

4. Train each model to predict future passenger counts.
5. Use Mean Squared Error (MSE) as the loss function and monitor the validation performance.
6. Evaluate the Models:
   a. Compare the performance of RNN, GRU, and LSTM on the test set using evaluation metrics like MSE and MAE.
   b. Plot the predicted vs. actual values to analyze model accuracy.
7. Compare the performance of the models for time series data.

This practical will strengthen your understanding of sequential neural networks and their applications in handling temporal data and natural language processing tasks.

10. **Attention-based Image Captioning**

In this practical, you will build an encoder-decoder model with an attention mechanism for generating image captions. The focus will be on understanding how to use pre-trained Convolutional Neural Networks (CNNs) as encoders and implementing attention mechanisms in decoders to highlight important image regions. This task will provide insights into the connection between vision and language tasks using deep learning models. You will work with the Flickr8k Image Caption Dataset. Steps to Follow

1. Load the Flickr8k Image Caption Dataset:
   a. Preprocess the dataset.
   b. Split the dataset into training, validation, and test sets.
2. Preprocess Text Data (Caption):
   a. Tokenize the captions and build a vocabulary to map words to indices.
   b. Pad sequences to ensure all captions are of equal length for training.
3. Load and Use Pre-trained CNNs as Encoders:
   a. Use pre-trained models like VGG16, ResNet, or InceptionV3 to extract image features.
   b. Pass the images through the CNN to obtain features that will serve as inputs to the decoder.
4. Define the Encoder:
   a. Use the pre-trained CNN as an encoder to extract features from images.
   b. Modify the CNN's output layer to match the size of the decoder's input.
5. Define the Decoder:
   a. Use an RNN-based decoder (e.g., LSTM or GRU) to generate captions from the image features.
   b. Implement attention mechanism to focus on relevant image regions during caption generation.
6. Attention Mechanism:
   a. Implement the Bahdanau or Luong attention mechanism:
      - Bahdanau: Calculates attention weights as a function of encoder-decoder hidden states.
      - Luong: Computes attention weights based on a weighted context vector derived from the encoder's output.
   b. Integrate the attention mechanism into the decoder to control the focus on different image regions when generating captions.
7. Training the Model:
   a. Train the encoder-decoder model using the Flickr8k dataset.
   b. Use cross-entropy loss to optimize the model's performance on caption prediction.
   c. Monitor training metrics such as loss and BLEU score on the validation set.

	8. Evaluate the Model: a. Generate captions for images in the test set and compare with ground truth captions. b. Calculate BLEU scores to evaluate the similarity between generated captions and the ground truth. This practical will enhance your ability to work with vision-language tasks and deepen your understanding of how attention mechanisms work in neural networks.

PRACTICAL : 1

AIM:

Solve the XOR Problem Using Multilayer Perceptrons (MLPs)

You are required to demonstrate how Multilayer Perceptrons (MLPs) solve non-linear problems like the XOR problem, showcasing their ability to handle complex decision boundaries. Follow these steps:

1. Design a Neural Network: Create a two-layer neural network with one hidden layer. Use a non-linear activation function (e.g., ReLU, sigmoid, or tanh) in the hidden layer.
2. Understand the Challenge: Begin by exploring why single-layer perceptrons cannot solve the XOR problem. Compare the limitations of single-layer perceptrons with the capabilities of MLPs.
3. Train the Network: Implement gradient descent to optimize the network's weights. Use appropriate loss and optimization functions to train the network on the XOR dataset.
4. Visualize the Decision Boundary: Once the network is trained, plot and interpret the decision boundary to understand how the MLP classifies the XOR problem.

This task will help you understand the importance of hidden layers and non-linear activation functions in solving non-linear problems.

CODE :-

```
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
from sklearn.metrics import accuracy_score
X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
y = np.array([[0], [1], [1], [0]])
```

```
model = tf.keras.Sequential([
    tf.keras.layers.Dense(4, input_dim=2, activation='tanh'),
    tf.keras.layers.Dense(1, activation='sigmoid')
])
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
history = model.fit(X, y, epochs=500, verbose=0)
y_pred = model.predict(X)
y_pred_labels = (y_pred > 0.5).astype(int)
accuracy = accuracy_score(y, y_pred_labels)
print(f"Training Accuracy: {accuracy:.2f}")
def plot_decision_boundary(model, X, y):
    x_min, x_max = X[:, 0].min() - 0.1, X[:, 0].max() + 0.1
    y_min, y_max = X[:, 1].min() - 0.1, X[:, 1].max() + 0.1
    xx, yy = np.meshgrid(np.linspace(x_min, x_max, 100),
                         np.linspace(y_min, y_max, 100))
    Z = model.predict(np.c_[xx.ravel(), yy.ravel()])
    Z = (Z > 0.5).astype(int).reshape(xx.shape)
    plt.contourf(xx, yy, Z, alpha=0.7)
    plt.scatter(X[:, 0], X[:, 1], c=y.flatten(), edgecolors='k')
    plt.title("Decision Boundary of XOR Problem")
    plt.xlabel("X1")
    plt.ylabel("X2")
    plt.show()
plot_decision_boundary(model, X, y)
```

OUTPUT:


/usr/local/lib/python3.11/dist-packages/keras/src/layers/core/dense.py:87: UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When using Sequential models, prefer using an `Input(shape)` object as the first la
super().__init__(activity_regularizer=activity_regularizer, **kwargs)
1/1 0s 90ms/step
Training Accuracy: 1.00
313/313 1s 2ms/step
Decision Boundary of XOR Problem
X1
X2
0.0
0.2
0.4
0.6
0.8
1.0

PRACTICAL : 2

AIM:

Implement and Compare Gradient Descent Variants

In this task, you will explore and compare the performance of different gradient descent optimizers. The objective is to understand how these variants affect training speed, convergence, and final accuracy. You will work with the MNIST Handwritten Digits dataset for this experiment. Follow these steps:

1. Choose Optimization Techniques: Implement the following gradient descent optimizers:
   a. Stochastic Gradient Descent (SGD)
   b. Momentum Gradient Descent (Momentum GD)
   c. RMSProp
   d. Adam
2. Train the Model: Use each optimizer to train a neural network on the MNIST dataset. Ensure you keep all other factors, such as the network architecture and initial learning rate, consistent for a fair comparison.
3. Analyze Performance:
   a. Compare the training speed of each optimizer by observing how quickly they reduce the loss.
   b. Analyze the convergence behavior to see which optimizer reaches the minimum loss most effectively.
   c. Evaluate the final accuracy of the model trained with each optimizer.
4. Hyperparameter Tuning: Experiment with hyperparameters (e.g., learning rates, decay rates) for each optimizer to observe how they influence performance.
5. Summarize Insights: Document the key differences among the optimizers and their suitability for various types of problems.

This task will help you understand the significance of choosing an appropriate optimization technique and how it impacts model performance and training efficiency.

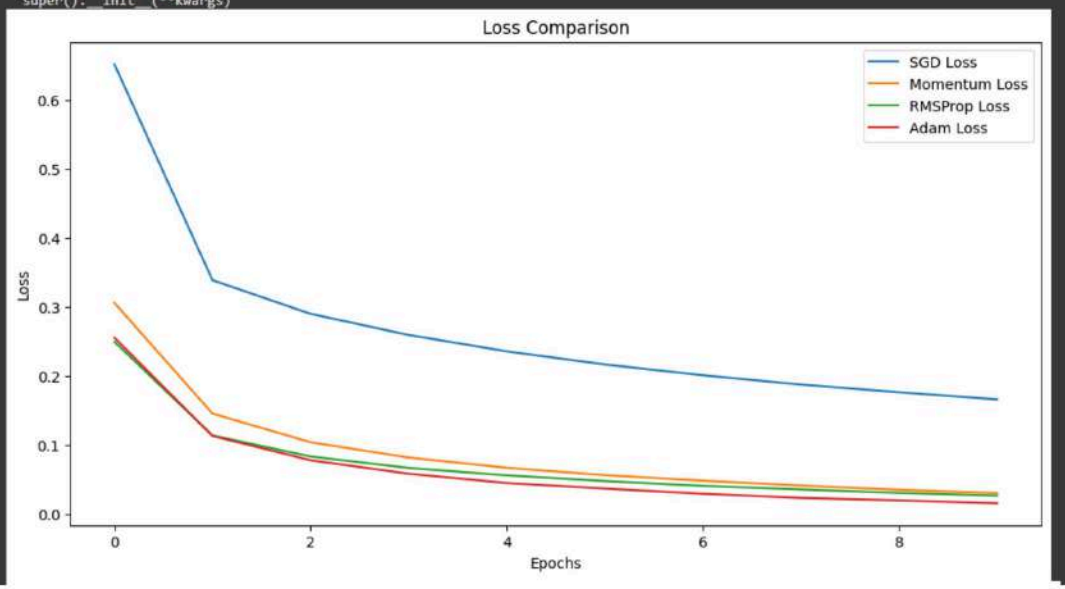
CODE :-

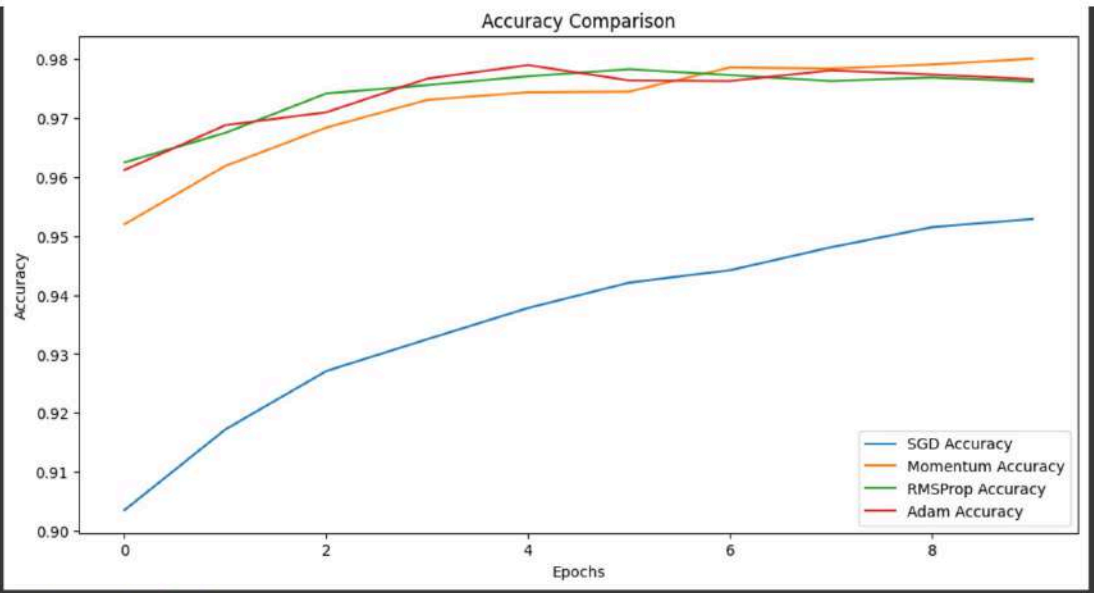
```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
from tensorflow.keras.datasets import mnist
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten
from tensorflow.keras.optimizers import SGD, RMSprop, Adam
(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0
optimizers = {
    "SGD": SGD(),
    "Momentum": SGD(momentum=0.9),
    "RMSProp": RMSprop(),
    "Adam": Adam()
}
def create_model():
    model = Sequential([
        Flatten(input_shape=(28, 28)),
        Dense(128, activation='relu'),
        Dense(10, activation='softmax')
    ])
    return model
history_dict = {}
```

```
for name, optimizer in optimizers.items():
    model = create_model()
    model.compile(optimizer=optimizer, loss='sparse_categorical_crossentropy', metrics=['accuracy'])
    history = model.fit(x_train, y_train, validation_data=(x_test, y_test), epochs=10, batch_size=32, verbose=0)
    history_dict[name] = history
plt.figure(figsize=(12, 6))
for name, history in history_dict.items():
    plt.plot(history.history['loss'], label=f'{name} Loss')
plt.title('Loss Comparison')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()
plt.figure(figsize=(12, 6))
for name, history in history_dict.items():
    plt.plot(history.history['val_accuracy'], label=f'{name} Accuracy')
plt.title('Accuracy Comparison')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
plt.show()
for name, history in history_dict.items():
    final_accuracy = history.history['val_accuracy'][-1]
    print(f'Final Accuracy with {name}: {final_accuracy:.4f}')
```

OUTPUT :-

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11490434/11490434 ━━━━━━━━━━━━━━━━━━━━ 0s 0us/step
/usr/local/lib/python3.11/dist-packages/keras/src/layers/reshaping/flatten.py:37: UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. Wh
  super().__init__(**kwargs)
```






```
Final Accuracy with SGD: 0.9529
Final Accuracy with Momentum: 0.9801
Final Accuracy with RMSProp: 0.9762
Final Accuracy with Adam: 0.9766
```

PRACTICAL : 3

AIM:

Implement the Backpropagation Algorithm

In this task, you will implement the backpropagation algorithm from scratch and understand how gradients are calculated for weight and bias updates in a neural network. You will work with a Simple Binary Classification Dataset (e.g., churn data converted to a binary problem). Follow these steps:

1. Set Up the Dataset: Prepare the churn dataset, ensuring it is formatted as a binary classification problem. Split the data into training and testing sets.
2. 2Design the Neural Network: Create a simple Multilayer Perceptron (MLP) with one hidden layer. Use appropriate activation functions for the hidden layer (e.g., ReLU, sigmoid, or tanh) and output layer (e.g., sigmoid for binary classification).
3. Implement Forward Propagation: Write the forward pass to calculate the outputs of the network layer by layer, including the final predicted outputs.
4. Implement Backpropagation:
   a. Derive the gradients of the loss function with respect to the weights and biases.
   b. Implement the chain rule to propagate errors backward through the network.
   c. Update the weights and biases using the calculated gradients and a chosen learning rate.
5. Train the Model: Use the implemented backpropagation algorithm to train the network on the training data. Monitor the loss at each epoch to ensure the model is learning.
6. Evaluate the Model: Test the model on the testing dataset and evaluate its performance using metrics such as accuracy, precision, and recall.
7. Summarize Insights: Reflect on how the errors propagate backward, how gradients are computed, and how these updates improve the model's performance.

This experiment will enhance your understanding of loss functions, activation functions, their gradients, and the essential role of backpropagation in training neural networks.

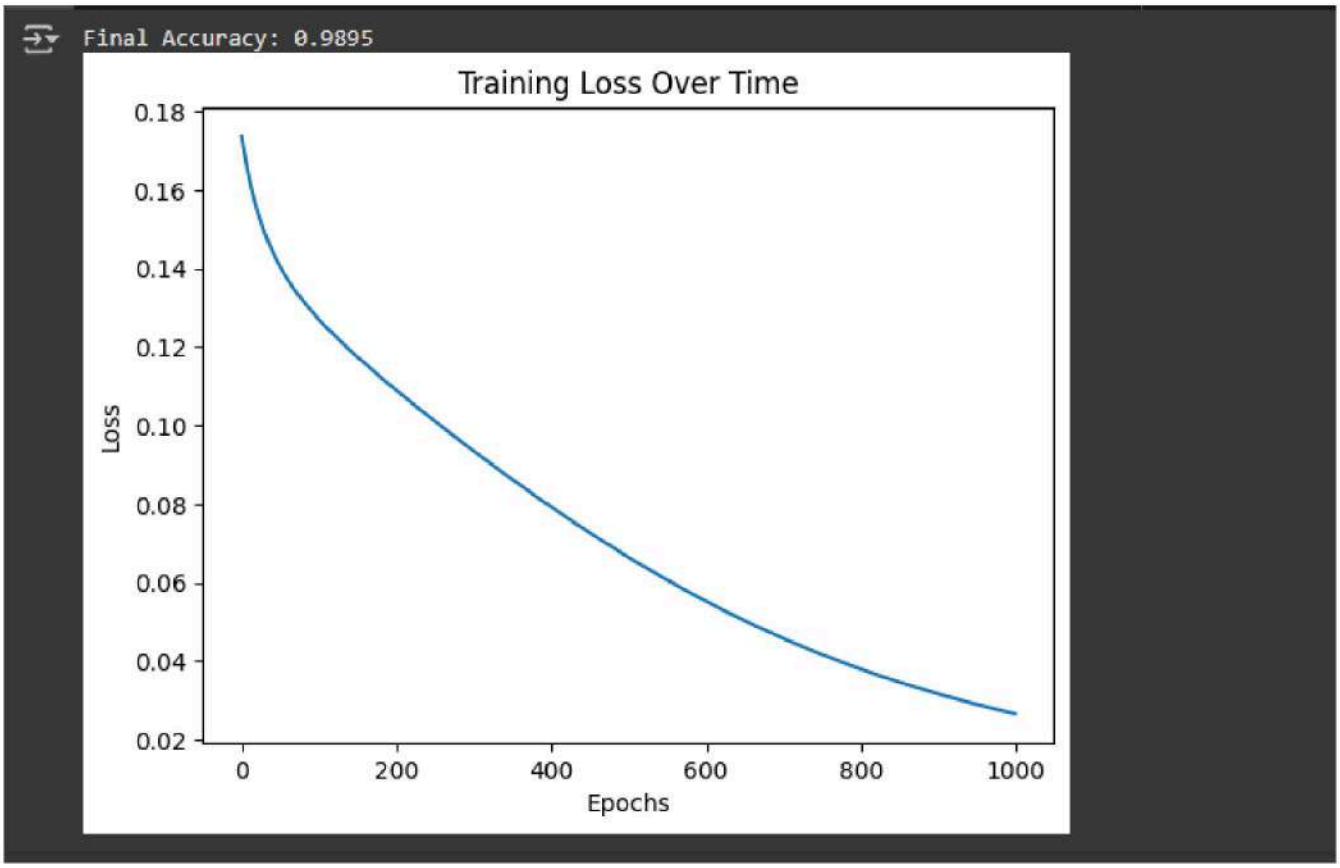
CODE :-

```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, LabelEncoder, OneHotEncoder
from sklearn.metrics import accuracy_score
import matplotlib.pyplot as plt
np.random.seed(42)
data = pd.read_csv('Churn_Modelling.csv')
label_encoder = LabelEncoder()
data['Gender'] = label_encoder.fit_transform(data['Gender'])
ohe = OneHotEncoder(drop='first', sparse_output=False)
geo_encoded = ohe.fit_transform(data[['Geography']])
geo_columns = [f'Geography_{cat}' for cat in ohe.categories_[0][1:]]
geo_df = pd.DataFrame(geo_encoded, columns=geo_columns)
data = pd.concat([data, geo_df], axis=1).drop(columns=['Geography'])
X = data.iloc[:, 3:-1].select_dtypes(include=[np.number]).values
y = data['Exited'].values.reshape(-1, 1)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
input_size = X_train.shape[1]
hidden_size = 4
output_size = 1
```

```
learning_rate = 0.1
epochs = 1000
W1 = np.random.randn(input_size, hidden_size)
b1 = np.zeros((1, hidden_size))
W2 = np.random.randn(hidden_size, output_size)
b2 = np.zeros((1, output_size))
def sigmoid(x):
   return 1 / (1 + np.exp(-x))
def sigmoid_derivative(x):
   return x * (1 - x)
loss_history = []
for epoch in range(epochs):
   Z1 = np.dot(X_train, W1) + b1
   A1 = sigmoid(Z1)
   Z2 = np.dot(A1, W2) + b2
   A2 = sigmoid(Z2)
      loss = np.mean((A2 - y_train) ** 2)
   loss_history.append(loss)
     dA2 = 2 * (A2 - y_train) / y_train.shape[0]
   dZ2 = dA2 * sigmoid_derivative(A2)
   dW2 = np.dot(A1.T, dZ2)
   db2 = np.sum(dZ2, axis=0, keepdims=True)
      dA1 = np.dot(dZ2, W2.T)
   dZ1 = dA1 * sigmoid_derivative(A1)
   dW1 = np.dot(X_train.T, dZ1)
   db1 = np.sum(dZ1, axis=0, keepdims=True)
      W1 -= learning_rate * dW1
   b1 -= learning_rate * db1
```

```
    W2 -= learning_rate * dW2
    b2 -= learning_rate * db2
Z1 = np.dot(X_test, W1) + b1
A1 = sigmoid(Z1)
Z2 = np.dot(A1, W2) + b2
A2 = sigmoid(Z2)
y_pred = (A2 > 0.5).astype(int)
accuracy = accuracy_score(y_test, y_pred)
print(f'Final Accuracy: {accuracy:.4f}')
plt.plot(loss_history)
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title('Training Loss Over Time')
plt.show()
```

OUTPUT :-



PRACTICAL : 4

AIM:

Explore the Impact of Regularization, Batch Normalization, and Weight Initialization on Model Training

In this task, you will explore how regularization, batch normalization, and advanced weight initialization techniques affect the training and generalization of deep networks. You will implement a Convolutional Neural Network (CNN) using any library and experiment with these methods on the CIFAR-10 Image Dataset. Steps to Follow

1. Set Up the Dataset:
   a. Load and preprocess the CIFAR-10 dataset.
   b. Normalize the image data and divide it into training, validation, and test sets.
2. Design the CNN:
   a. Build a CNN architecture suitable for the CIFAR-10 dataset.
   b. Ensure the model includes convolutional, pooling, and fully connected layers.
3. Experiment with Regularization:
   a. Apply techniques like L1/L2 regularization (weight decay) and Dropout.
   b. Train the model with and without regularization, and compare the impact on overfitting and generalization.
4. Add Batch Normalization:
   a. Integrate batch normalization layers after the convolutional and fully connected layers.
   b. Train the model with and without batch normalization, and observe its effect on training stability, convergence speed, and accuracy.
5. Experiment with Weight Initialization:
   a. Use different weight initialization strategies, such as Random Initialization, Xavier Initialization, and He Initialization.
   b. Compare their effects on training, focusing on addressing vanishing and

exploding gradients.

6. Train and Compare Models:

   a. Train the models with various combinations of regularization, batch normalization, and weight initialization.

   b. Monitor training and validation performance (accuracy and loss) to evaluate generalization.

7. Analyze Results:

   a. Compare the models' performance based on training stability, convergence speed, and accuracy.

   b. Highlight how each technique contributes to improved model performance.

8. Summarize Insights: Reflect on the role of regularization in preventing overfitting. Discuss how batch normalization stabilizes training and accelerates convergence. Explain the importance of proper weight initialization in avoiding gradient-related issues.

This practical will help you understand critical techniques that improve training and generalization in deep networks, preparing you for advanced model design and optimization.

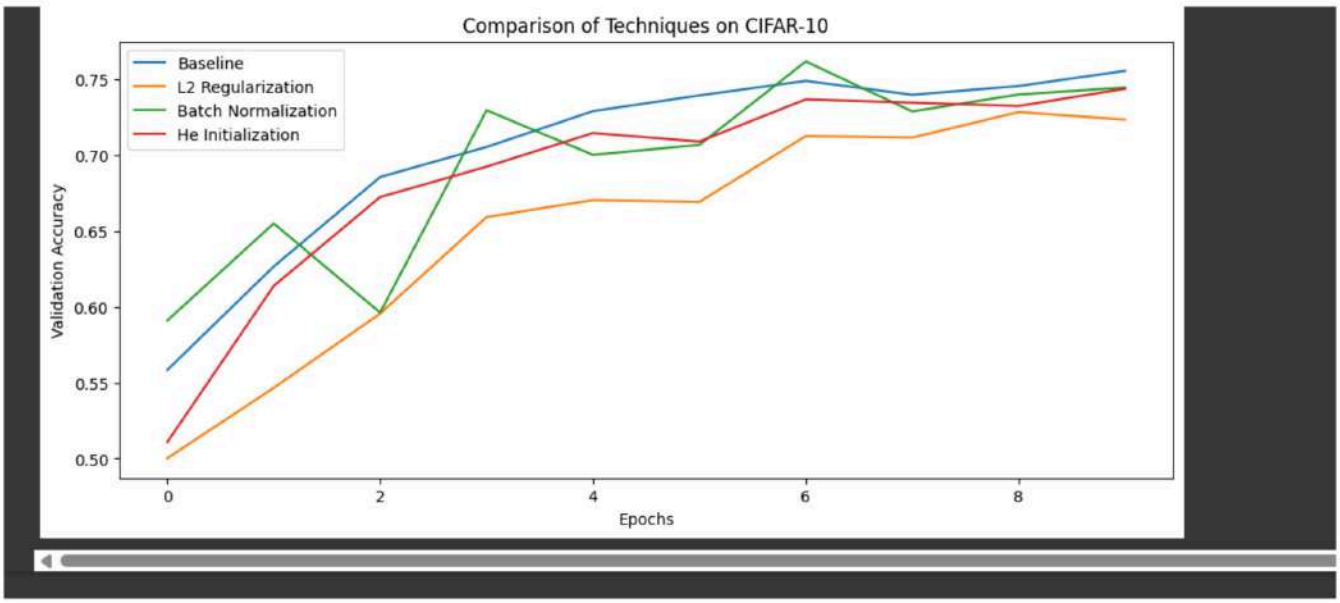
CODE :-

```
import tensorflow as tf
from tensorflow.keras import layers, models, regularizers, optimizers, initializers
import matplotlib.pyplot as plt
import numpy as np
from tensorflow.keras.datasets import cifar10
# Load and preprocess the CIFAR-10 dataset
(x_train, y_train), (x_test, y_test) = cifar10.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0  # Normalize images
y_train, y_test = tf.keras.utils.to_categorical(y_train, 10), tf.keras.utils.to_categorical(y_test, 10)
```

```
# Split validation set from training data
x_train, x_val = x_train[:45000], x_train[45000:]
y_train, y_val = y_train[:45000], y_train[45000:]
# Function to build CNN model with optional regularization and batch normalization
def build_model(reg_type=None, batch_norm=False, init_type='glorot_uniform'):
    model = models.Sequential()
        # Convolutional layers
    model.add(layers.Conv2D(32, (3, 3), padding='same', activation='relu',
                            kernel_initializer=init_type, kernel_regularizer=reg_type, input_shape=(32, 32, 3)))
    if batch_norm:
        model.add(layers.BatchNormalization())
    model.add(layers.MaxPooling2D((2, 2)))
        model.add(layers.Conv2D(64, (3, 3), padding='same', activation='relu',
                            kernel_initializer=init_type, kernel_regularizer=reg_type))
    if batch_norm:
        model.add(layers.BatchNormalization())
    model.add(layers.MaxPooling2D((2, 2)))
        model.add(layers.Conv2D(128, (3, 3), padding='same', activation='relu',
                            kernel_initializer=init_type, kernel_regularizer=reg_type))
    if batch_norm:
        model.add(layers.BatchNormalization())
    model.add(layers.MaxPooling2D((2, 2)))
        # Fully connected layers
    model.add(layers.Flatten())
    model.add(layers.Dense(128, activation='relu', kernel_initializer=init_type,
kernel_regularizer=reg_type))
    if batch_norm:
        model.add(layers.BatchNormalization())
```

```
    model.add(layers.Dropout(0.5))
        model.add(layers.Dense(10, activation='softmax'))
    return model
# Experimenting with different configurations
configs = [
    {'reg': None, 'batch_norm': False, 'init': 'glorot_uniform', 'name': 'Baseline'},
    {'reg': regularizers.l2(0.001), 'batch_norm': False, 'init': 'glorot_uniform', 'name': 'L2 Regularization'},
    {'reg': None, 'batch_norm': True, 'init': 'glorot_uniform', 'name': 'Batch Normalization'},
    {'reg': None, 'batch_norm': False, 'init': 'he_normal', 'name': 'He Initialization'},
]
histories = {}
for config in configs:
    print(f"Training model: {config['name']}")
    model = build_model(reg_type=config['reg'], batch_norm=config['batch_norm'], init_type=config['init'])
    model.compile(optimizer=optimizers.Adam(), loss='categorical_crossentropy', metrics=['accuracy'])
    history = model.fit(x_train, y_train, epochs=10, validation_data=(x_val, y_val), batch_size=64, verbose=1)
    histories[config['name']] = history
# Plotting the training loss and accuracy
plt.figure(figsize=(12, 5))
for name, history in histories.items():
    plt.plot(history.history['val_accuracy'], label=name)
plt.xlabel('Epochs')
plt.ylabel('Validation Accuracy')
plt.title('Comparison of Techniques on CIFAR-10')
plt.legend()
plt.show()
```

OUTPUT :-


Comparison of Techniques on CIFAR-10
Baseline
L2 Regularization
Batch Normalization
He Initialization
Validation Accuracy
0.75
0.70
0.65
0.60
0.55
0.50
0
2
4
6
8
Epochs

PRACTICAL : 5

AIM:

Dimensionality Reduction Using PCA

In this task, you will apply Principal Component Analysis (PCA) to perform dimensionality reduction and visualize high-dimensional data in 2D space. You will work with the Wine Dataset to understand how PCA extracts principal components and their significance in explaining data variance. Steps to Follow

1. Set Up the Dataset:

   a. Load the Wine dataset and preprocess it by normalizing or standardizing the features to ensure they are on the same scale.

2. Apply PCA:

   a. Compute the principal components of the dataset.

   b. Reduce the dimensionality of the data to 2 components and visualize it in a 2D scatter plot, labeling the data points by their class.

3. Analyze Principal Components:

   a. Examine the explained variance ratio of each principal component to understand how much of the data variance is captured.

4. Reconstruct Data:

   a. Reconstruct the dataset from the reduced dimensions and compare it with the original data.

   b. Quantify the reconstruction error to observe the trade-off between dimensionality reduction and information loss.

5. Visualize Results:

   a. Plot the original data (if feasible) and the reduced data to visually interpret the difference.

   b. Highlight how PCA separates the classes in the Wine dataset based on its variance-capturing property.

6. Summarize Insights: Reflect on how PCA helps in reducing the complexity of high-dimensional data while retaining most of the variance. Discuss the balance between reducing dimensions and preserving important information.

This experiment will give you practical insights into the use of PCA for dimensionality reduction, its effect on data interpretation, and its importance in simplifying high-dimensional datasets.

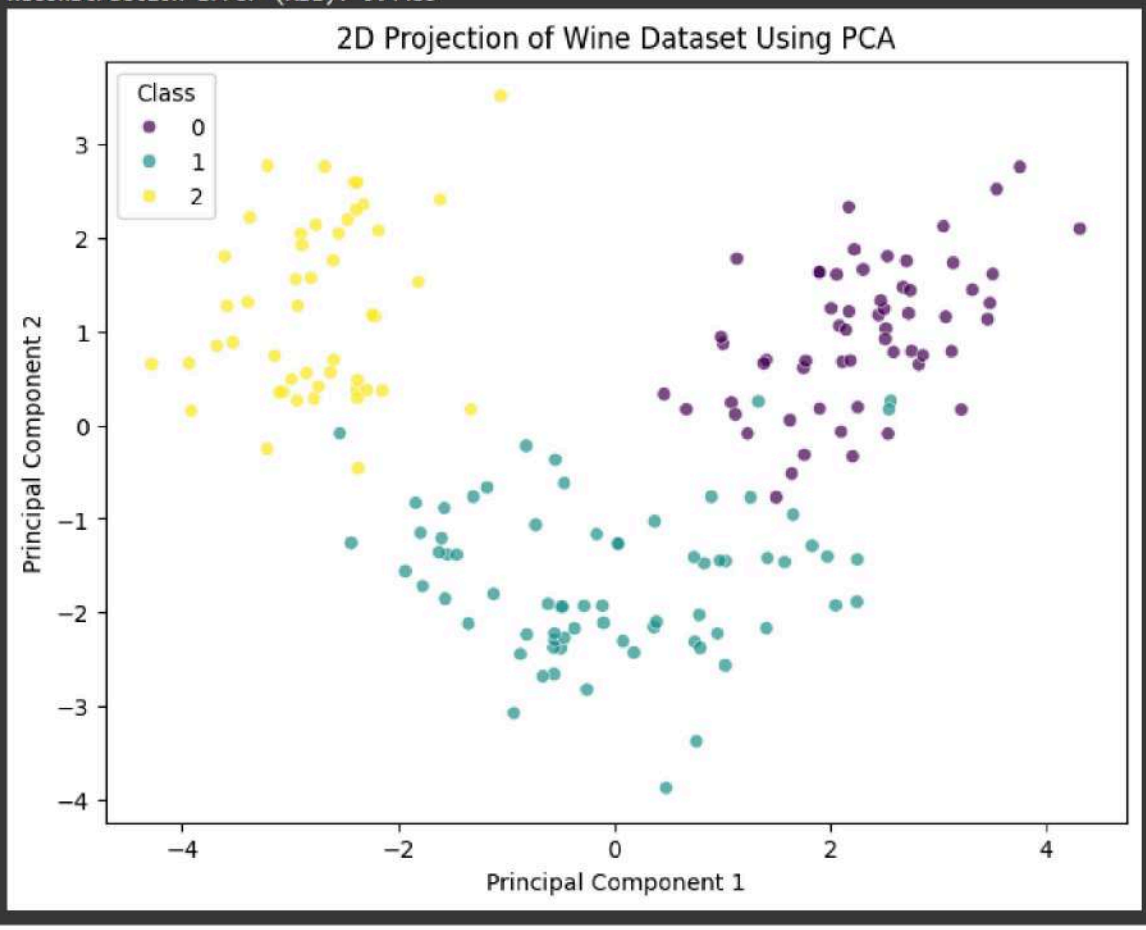
CODE :-

```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler
from sklearn.datasets import load_wine
from sklearn.metrics import mean_squared_error
# Step 1: Load and Preprocess the Dataset
wine = load_wine()
X = wine.data
y = wine.target
# Standardizing the features
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
# Step 2: Apply PCA
pca = PCA(n_components=2)
X_pca = pca.fit_transform(X_scaled)
# Step 3: Analyze Principal Components
explained_variance = pca.explained_variance_ratio_
print(f"Explained Variance Ratios: {explained_variance}")
```

```
# Step 4: Reconstruct Data

X_reconstructed = pca.inverse_transform(X_pca)

reconstruction_error = mean_squared_error(X_scaled, X_reconstructed)

print(f"Reconstruction Error (MSE): {reconstruction_error:.4f}")

# Step 5: Visualize Results

plt.figure(figsize=(8, 6))

sns.scatterplot(x=X_pca[:, 0], y=X_pca[:, 1], hue=y, palette='viridis', alpha=0.7)

plt.xlabel("Principal Component 1")

plt.ylabel("Principal Component 2")

plt.title("2D Projection of Wine Dataset Using PCA")

plt.legend(title="Class")

plt.show()
```

OUTPUT :-

```
Explained Variance Ratios: [0.36198848 0.1920749 ]
Reconstruction Error (MSE): 0.4459
```



PRACTICAL : 6

AIM:

Denoising Images Using Autoencoders

In this task, you will implement a denoising autoencoder to clean noisy images, using the MNIST Handwritten Digits dataset. You will explore the encoding, decoding, and reconstruction processes in neural networks and understand how autoencoders relate to PCA. This experiment will also highlight the role of regularization and noise handling in autoencoders. Steps to Follow

1. Set Up the Dataset:

   a. Load and preprocess the MNIST dataset.

   b. Introduce noise to the images by adding random Gaussian or salt-and-pepper noise. Retain the original images as clean versions for comparison.

2. Design the Autoencoder:

   a. Build a neural network architecture with an encoder that compresses the input images into a lower-dimensional latent space and a decoder that reconstructs the images from the latent representation.

   b. Use non-linear activation functions like ReLU or sigmoid for the layers.

3. Train the Autoencoder:

   a. Train the autoencoder with noisy images as input and clean images as output.

   b. Use a suitable loss function, such as Mean Squared Error (MSE), to measure the difference between the reconstructed and original images.

   c. Monitor training loss to ensure the model is learning to remove noise effectively.

4. Test the Autoencoder:

   a. Evaluate the model on a separate test set of noisy images and compare the reconstructed images with their clean counterparts.

5. Analyze Regularization:

a. Experiment with adding regularization techniques, such as L1/L2 penalties or Dropout, to the autoencoder.

b. Observe their impact on the reconstruction quality and model generalization.

6. Visualize Results:

a. Display side-by-side comparisons of noisy, clean, and reconstructed images.

b. Plot the latent space representation to understand how the autoencoder encodes the input data.

7. Summarize Insights: Reflect on how denoising autoencoders handle noise effectively and improve image quality. Discuss the differences between autoencoders and PCA, emphasizing the power of neural networks in capturing non-linear features.

This practical will enhance your understanding of autoencoders, their applications in denoising.

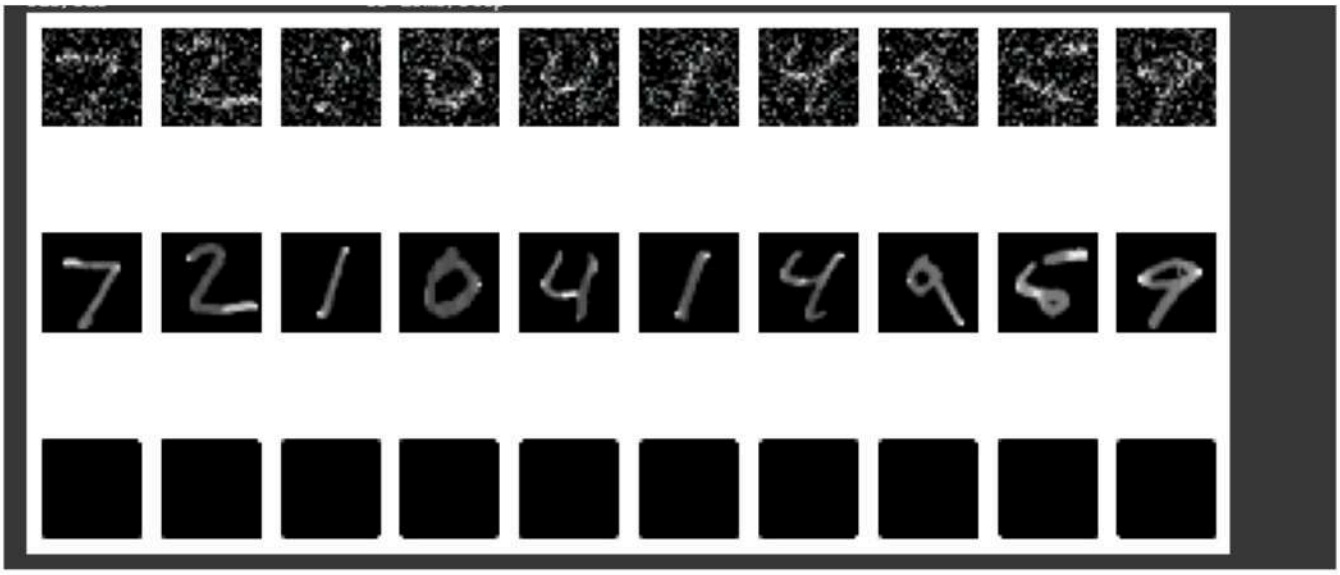
CODE :-

```
import numpy as np
import matplotlib.pyplot as plt
from tensorflow.keras.datasets import mnist
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, Conv2D, Conv2DTranspose, MaxPooling2D, UpSampling2D
from tensorflow.keras.utils import normalize
from tensorflow.keras import regularizers
# Load and preprocess the MNIST dataset
(x_train, y_train), (x_test, y_test) = mnist.load_data()
# Normalize and reshape
x_train = normalize(x_train.astype('float32'), axis=1)
x_test = normalize(x_test.astype('float32'), axis=1)
x_train = np.expand_dims(x_train, axis=-1)
x_test = np.expand_dims(x_test, axis=-1)
```

```
# Add Gaussian noise
def add_noise(images, noise_factor=0.3):
    noise = np.random.normal(loc=0.0, scale=noise_factor, size=images.shape)
    noisy_images = np.clip(images + noise, 0., 1.)
    return noisy_images
x_train_noisy = add_noise(x_train)
x_test_noisy = add_noise(x_test)
# Build the Autoencoder
input_img = Input(shape=(28, 28, 1))
# Encoder
x = Conv2D(32, (3, 3), activation='relu', padding='same')(input_img)
x = MaxPooling2D((2, 2), padding='same')(x)
x = Conv2D(64, (3, 3), activation='relu', padding='same')(x)
x = MaxPooling2D((2, 2), padding='same')(x)
# Decoder
x = Conv2DTranspose(64, (3, 3), activation='relu', padding='same')(x)
x = UpSampling2D((2, 2))(x)
x = Conv2DTranspose(32, (3, 3), activation='relu', padding='same')(x)
x = UpSampling2D((2, 2))(x)
# Output layer
output_img = Conv2D(1, (3, 3), activation='sigmoid', padding='same')(x)
# Compile and train the model
autoencoder = Model(input_img, output_img)
autoencoder.compile(optimizer='adam', loss='mean_squared_error')
# Training the model
autoencoder.fit(x_train_noisy, x_train, epochs=10, batch_size=256, validation_data=(x_test_noisy,
x_test))
# Denoising the images
```

```
denoised_images = autoencoder.predict(x_test_noisy)

# Visualize results

plt.figure(figsize=(10, 5))

for i in range(10):

    # Noisy images

    ax = plt.subplot(3, 10, i + 1)

    plt.imshow(x_test_noisy[i].reshape(28, 28), cmap='gray')

    ax.axis('off')

    # Clean images

    ax = plt.subplot(3, 10, i + 11)

    plt.imshow(x_test[i].reshape(28, 28), cmap='gray')

    ax.axis('off')

    # Denoised images

    ax = plt.subplot(3, 10, i + 21)

    plt.imshow(denoised_images[i].reshape(28, 28), cmap='gray')

    ax.axis('off')

plt.show()
```

OUTPUT :-



PRACTICAL : 7

AIM:

Visualizing CNN Filters and Feature Maps

In this task, you will explore Convolutional Neural Networks (CNNs) by visualizing the filters they learn and the feature maps generated during feature extraction. This will help you understand the hierarchical learning process in CNNs. You will work with two datasets and tasks:

Part1: CIFAR-10 Dataset: Train a CNN from scratch to visualize filters and feature maps.

Part2: Chest X-ray Pneumonia Dataset: Fine-tune a pretrained CNN (e.g., ResNet50, VGG16, or EfficientNet) for medical image classification and interpret the feature extraction process.

Part 1: Visualizing CNN Filters and Feature Maps (CIFAR-10)

1. Set Up the Dataset:
   a. Load and preprocess the CIFAR-10 dataset (e.g., normalize images and split into training, validation, and test sets).
2. Train a CNN:
   a. Build a CNN architecture from scratch or use a simple predefined model.
   b. Train the CNN on CIFAR-10 and evaluate its performance on the validation set.
3. Visualize Filters:
   a. Extract and visualize the learned filters from the convolutional layers.
   b. Understand how these filters capture basic patterns like edges, corners, and textures.
4. Inspect Feature Maps:
   a. For a given input image, pass it through the network and capture intermediate feature maps.
   b. Visualize these feature maps to observe how the network extracts hierarchical features at different layers.

5. Analyze Results: Discuss the progression from low-level features (edges and textures) in early layers to high-level features (shapes and objects) in deeper layers.

Part 2: Fine-Tuning Pretrained CNN Models (Chest X-ray Pneumonia Dataset)

1. Set Up the Dataset:

   a. Load the Chest X-ray Pneumonia dataset and preprocess it (resize images, normalize pixel values, and split into training, validation, and test sets).

2. Choose a Pretrained Model:

   a. Select a pretrained CNN model such as ResNet50, VGG16, or EfficientNet.

   b. Fine-tune the model by replacing the final classification layer with a layer suitable for binary classification (pneumonia vs. normal).

3. Train the Model:

   a. Train the model on the Chest X-ray dataset and evaluate its performance using metrics like accuracy, precision, and recall.

4. Visualize Filters and Feature Maps:

   a. Visualize the filters learned in the convolutional layers of the pretrained model.

   b. Pass a sample chest X-ray image through the network and capture intermediate feature maps to observe feature extraction at different stages.

5. Analyze Results: Compare the filters and feature maps of the pretrained model with those of the scratch-trained CIFAR-10 model.

6. Discuss how pretrained models adapt to specific tasks like medical image classification.

This practical will deepen your understanding of CNNs, hierarchical feature learning, and the power of transfer learning in specialized tasks.

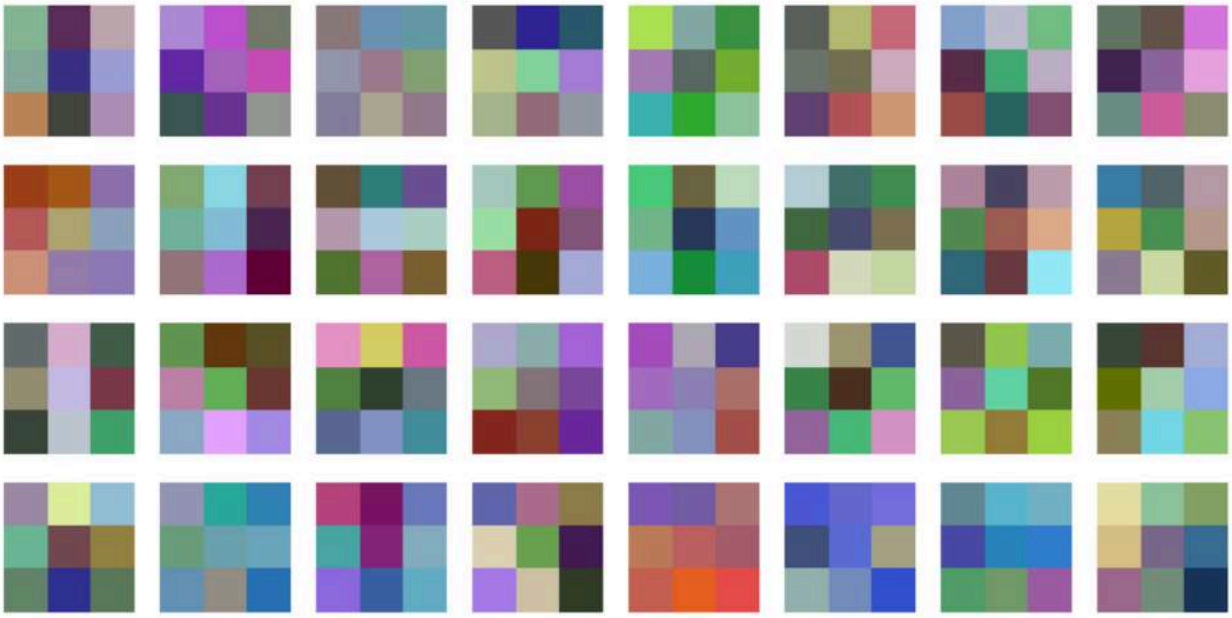
CODE :-

```
import tensorflow as tf
from tensorflow import keras
import numpy as np
import matplotlib.pyplot as plt


# Load and preprocess CIFAR-10 dataset
(x_train, y_train), (x_test, y_test) = keras.datasets.cifar10.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0


# Define a simple CNN model
def create_cnn():
    model = keras.Sequential([
        keras.layers.Conv2D(32, (3,3), activation='relu', input_shape=(32,32,3)),
        keras.layers.MaxPooling2D(2,2),
        keras.layers.Conv2D(64, (3,3), activation='relu'),
        keras.layers.MaxPooling2D(2,2),
        keras.layers.Flatten(),
        keras.layers.Dense(128, activation='relu'),
        keras.layers.Dense(10, activation='softmax')
    ])
    return model


# Compile and train the model
cnn_model = create_cnn()
cnn_model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])
cnn_model.fit(x_train, y_train, epochs=10, validation_data=(x_test, y_test), batch_size=64)
```

```
# Visualize Filters from the first Conv layer
layer = cnn_model.layers[0]
filters, biases = layer.get_weights()
filters = (filters - filters.min()) / (filters.max() - filters.min())
fig, axes = plt.subplots(4, 8, figsize=(10,5))
for i, ax in enumerate(axes.flat):
    if i < 32:
        ax.imshow(filters[:, :, :, i], cmap='gray')
        ax.axis('off')
plt.show()

# Visualizing Feature Maps
layer_outputs = [layer.output for layer in cnn_model.layers if 'conv' in layer.name]
feature_map_model = keras.Model(inputs=cnn_model.input, outputs=layer_outputs)
sample_image = np.expand_dims(x_test[0], axis=0)
feature_maps = feature_map_model.predict(sample_image)
fig, axes = plt.subplots(4, 8, figsize=(10,5))
for i, ax in enumerate(axes.flat):
    if i < 32:
        ax.imshow(feature_maps[0][:, :, i], cmap='gray')
        ax.axis('off')
plt.show()

# Part 2: Fine-tuning a pretrained model on Chest X-ray dataset
base_model = keras.applications.VGG16(weights='imagenet', include_top=False, input_shape=(224, 224, 3))
for layer in base_model.layers:
    layer.trainable = False
```

```
# Add classification layers
x = keras.layers.Flatten()(base_model.output)
x = keras.layers.Dense(256, activation='relu')(x)
x = keras.layers.Dense(1, activation='sigmoid')(x)
fine_tuned_model = keras.Model(inputs=base_model.input, outputs=x)

# Compile the model
fine_tuned_model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])

# Print summary to verify architecture
fine_tuned_model.summary()
```

OUTPUT :-



PRACTICAL : 8

AIM:

Text Pre-Processing and Word Embedding

In this task, you will prepare textual data for machine learning models by applying text pre-processing techniques and converting text into numerical vectors using word embedding methods. The goal is to understand how pre-processing cleans and standardizes text and how embeddings like GloVe, Word2Vec, and BERT capture semantic relationships in text data. You will use the IMDB Movie Reviews Dataset for a sentiment analysis task. Steps to Follow

1. Set Up the Dataset:

   a. Load the IMDB Movie Reviews dataset and split it into training and test sets.

2. Perform Text Pre-Processing:

   a. Apply the following steps to clean and prepare the text:

   i. Convert text to lowercase.

   ii. Remove punctuation, numbers, and special characters.

   iii. Tokenize the text into words.

   iv. Remove stopwords using libraries like NLTK or Spacy.

   v. Apply stemming or lemmatization to normalize words.

3. Implement Word Embedding Techniques:

4. Static Embeddings (GloVe and Word2Vec):

   a. Use pre-trained GloVe or Word2Vec models from Gensim to convert each word into a fixed-length dense vector.

   b. Represent each review as an average or sum of its word vectors.

4. Dynamic Embeddings (BERT):

   a. Use the Hugging Face Transformers library to extract contextual embeddings from BERT.

   b. Tokenize the text using BERT's tokenizer and represent each review by the

embedding of the [CLS] token or the average of token embeddings.

5. Train a Sentiment Analysis Model:

   a. Use the generated embeddings as input features for a classifier (e.g., logistic regression, random forest, or a simple neural network).

   b. Train separate models for GloVe, Word2Vec, and BERT embeddings to compare their performance.

6. Evaluate Model Performance:

   a. Evaluate the models on the test set using accuracy, precision, recall, and F1-score.

   b. Analyze the impact of different embeddings on the classifier's performance.

7. Analyze and Compare Embeddings:

   a. Highlight the key differences between static (GloVe and Word2Vec) and dynamic embeddings (BERT).

   b. Discuss how BERT considers the context of words, while GloVe and Word2Vec produce the same vector for a word regardless of its context.

8. Summarize Insights: Reflect on the importance of text pre-processing in cleaning and standardizing textual data. Discuss how word embeddings enhance feature representation by capturing semantic meanings. Highlight the strengths of dynamic embeddings like BERT in handling context compared to static embeddings like GloVe and Word2Vec.

This practical will deepen your understanding of text processing, word embeddings, and their application in Natural Language Processing (NLP) tasks.

CODE :-

```
import numpy as np
import tensorflow as tf
import nltk
import re
```

```
import gensim.downloader as api
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
from transformers import BertTokenizer, TFBertModel
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score

nltk.download('punkt')
nltk.download('stopwords')

# Load IMDB dataset
imdb = tf.keras.datasets.imdb
(x_train, y_train), (x_test, y_test) = imdb.load_data()
word_index = imdb.get_word_index()
reverse_word_index = {value: key for key, value in word_index.items()}

# Decode IMDB reviews
def decode_review(encoded_review):
    return ' '.join([reverse_word_index.get(i - 3, '?') for i in encoded_review])

x_train = [decode_review(review) for review in x_train[:5000]]  # Limit for memory efficiency
x_test = [decode_review(review) for review in x_test[:2000]]

# Preprocessing function
def preprocess_text(text):
```

```
    text = text.lower()
    text = re.sub(r'[^a-zA-Z\s]', '', text)
    tokens = word_tokenize(text)
    tokens = [word for word in tokens if word not in stopwords.words('english')]
    return ' '.join(tokens)

x_train = [preprocess_text(review) for review in x_train]
x_test = [preprocess_text(review) for review in x_test]

# Load pre-trained GloVe model (handling micropip issue)
try:
    glove_model = api.load("glove-wiki-gigaword-50")
except ModuleNotFoundError:
    print("Error: gensim requires micropip. Ensure the environment supports external downloads.")
    glove_model = None

def get_glove_embedding(text):
    if glove_model is None:
        return np.zeros(50)
    words = text.split()
    word_vectors = [glove_model[word] for word in words if word in glove_model]
    return np.mean(word_vectors, axis=0) if word_vectors else np.zeros(50)

x_train_glove = np.array([get_glove_embedding(review) for review in x_train])
x_test_glove = np.array([get_glove_embedding(review) for review in x_test])

# Train classifier using GloVe embeddings
clf = LogisticRegression()
```

```
clf.fit(x_train_glove, y_train[:5000])

y_pred = clf.predict(x_test_glove)


# Evaluate model

print("Accuracy:", accuracy_score(y_test[:2000], y_pred))

print("Precision:", precision_score(y_test[:2000], y_pred))

print("Recall:", recall_score(y_test[:2000], y_pred))

print("F1-score:", f1_score(y_test[:2000], y_pred))


# Load pre-trained BERT model

tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')

bert_model = TFBertModel.from_pretrained('bert-base-uncased')


def get_bert_embedding(text):

    tokens = tokenizer(text, return_tensors='tf', padding=True, truncation=True, max_length=512)

    outputs = bert_model(**tokens)

    return outputs.last_hidden_state[:, 0, :].numpy().flatten()


x_train_bert = np.array([get_bert_embedding(review) for review in x_train[:1000]])

x_test_bert = np.array([get_bert_embedding(review) for review in x_test[:1000]])

y_train_bert = y_train[:1000]

y_test_bert = y_test[:1000]


# Train classifier using BERT embeddings

clf_bert = LogisticRegression()

clf_bert.fit(x_train_bert, y_train_bert)

y_pred_bert = clf_bert.predict(x_test_bert)
```

```
# Evaluate BERT model

print("BERT Accuracy:", accuracy_score(y_test_bert, y_pred_bert))

print("BERT Precision:", precision_score(y_test_bert, y_pred_bert))

print("BERT Recall:", recall_score(y_test_bert, y_pred_bert))

print("BERT F1-score:", f1_score(y_test_bert, y_pred_bert))
```

OUTPUT :-

```
[nltk_data] Downloading package punkt to /root/nltk_data...
[nltk_data]   Unzipping tokenizers/punkt.zip.
[nltk_data] Downloading package stopwords to /root/nltk_data...
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/imdb.npz
      0/17464789 ━━━━━━━━━━━━━━━━━━━━ 0s 0s/step[nltk_data]   Unzipping corpora/stopwords.zip.
17464789/17464789 ━━━━━━━━━━━━━━━━━━━━ 1s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/imdb_word_index.json
1641221/1641221 ━━━━━━━━━━━━━━━━━━━━ 1s 0us/step
```

PRACTICAL : 9

AIM:

Implementation of Sequential Models

In this task, you will explore sequential data processing by implementing Recurrent Neural Networks (RNNs), Gated Recurrent Units (GRUs), and Long Short-Term Memory (LSTM) networks. These models are foundational for capturing temporal dependencies in sequential data. You will address challenges like vanishing gradients in RNNs and see how GRUs and LSTMs overcome them using gating mechanisms. The task involves training and evaluating these models on two datasets for different sequential tasks: Steps to Follow

1. Load and Preprocess the Dataset:

   a. Load the Airline Passenger dataset (e.g., monthly passenger counts).

   b. Normalize the data for efficient training.

   c. Prepare the data for sequence modeling by creating sliding windows of input-output pairs.

2. Implement Sequential Models:

   a. Build and train the following sequential models:

   - RNN: A basic recurrent neural network to learn temporal dependencies.
   - GRU: A gated model addressing RNN limitations by selectively retaining information.
   - LSTM: A model with forget and input gates to capture long-term dependencies effectively.

3. Train the Models:
4. Train each model to predict future passenger counts.
5. Use Mean Squared Error (MSE) as the loss function and monitor the validation performance.
6. Evaluate the Models:

   a. Compare the performance of RNN, GRU, and LSTM on the test set using

evaluation metrics like MSE and MAE.

b. Plot the predicted vs. actual values to analyze model accuracy.

7. Compare the performance of the models for time series data.

This practical will strengthen your understanding of sequential neural networks and their applications in handling temporal data and natural language processing tasks.

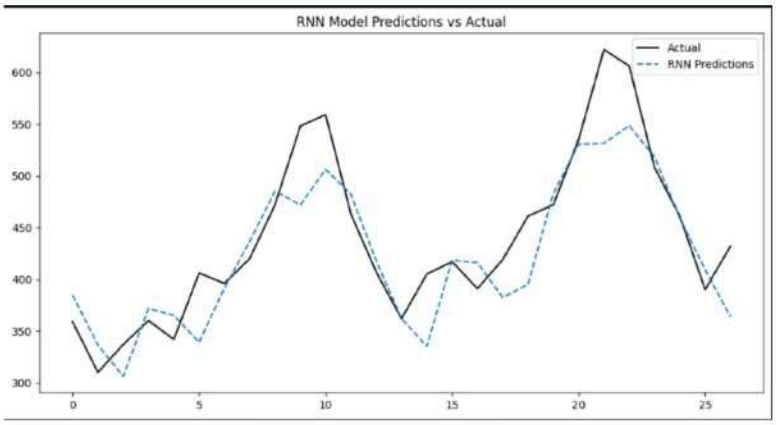
CODE :-

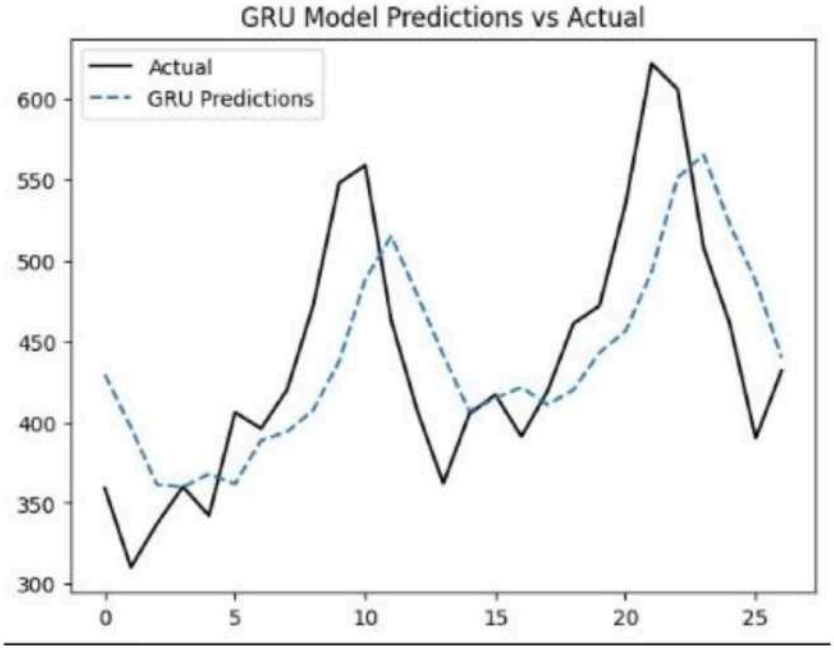
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import SimpleRNN, GRU, LSTM, Dense
from sklearn.preprocessing import MinMaxScaler

# Step 1: Load and Preprocess the Dataset
df = pd.read_csv("AirPassengers.csv")  # Ensure the dataset is available
df['Month'] = pd.to_datetime(df['Month'])
df.set_index('Month', inplace=True)

data = df['#Passengers'].values.reshape(-1, 1)
scaler = MinMaxScaler(feature_range=(0, 1))
data_scaled = scaler.fit_transform(data)

# Function to create sliding windows
def create_sequences(data, seq_length):
    X, y = [], []
    for i in range(len(data) - seq_length):
```

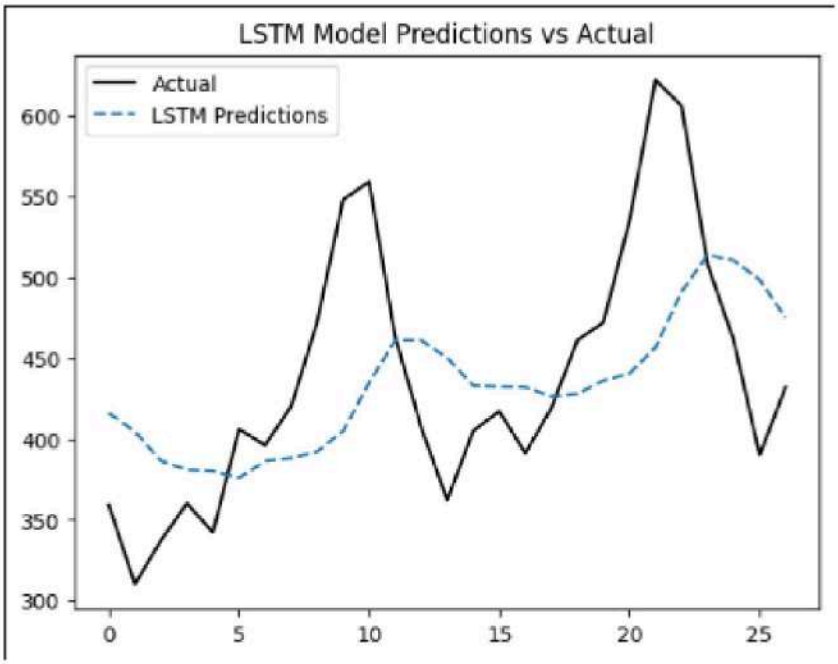
```
        X.append(data[i:i + seq_length])
        y.append(data[i + seq_length])
    return np.array(X), np.array(y)

seq_length = 10  # Define the sequence length
X, y = create_sequences(data_scaled, seq_length)
X_train, X_test = X[:int(len(X) * 0.8)], X[int(len(X) * 0.8):]
y_train, y_test = y[:int(len(y) * 0.8)], y[int(len(y) * 0.8):]

# Step 2: Define Sequential Models
def build_model(model_type):
    model = Sequential()
    if model_type == "RNN":
        model.add(SimpleRNN(50, activation='tanh', return_sequences=False, input_shape=(seq_length,
1)))
    elif model_type == "GRU":
        model.add(GRU(50, activation='tanh', return_sequences=False, input_shape=(seq_length, 1)))
    elif model_type == "LSTM":
        model.add(LSTM(50, activation='tanh', return_sequences=False, input_shape=(seq_length, 1)))

    model.add(Dense(1))
    model.compile(optimizer='adam', loss='mse', metrics=['mae'])
    return model

# Step 3: Train and Evaluate Models
models = {}
for model_type in ["RNN", "GRU", "LSTM"]:
    print(f"Training {model_type}...")
```

```
    model = build_model(model_type)
    model.fit(X_train, y_train, epochs=50, batch_size=16, validation_data=(X_test, y_test), verbose=1)
    models[model_type] = model

# Step 4: Evaluate Models
def evaluate_models(models):
    plt.figure(figsize=(12, 6))
    for model_type, model in models.items():
        predictions = model.predict(X_test)
        predictions = scaler.inverse_transform(predictions)
        actual = scaler.inverse_transform(y_test.reshape(-1, 1))

        plt.plot(actual, label=f"Actual", color='black')
        plt.plot(predictions, label=f"{model_type} Predictions", linestyle='dashed')
        plt.title(f"{model_type} Model Predictions vs Actual")
        plt.legend()
        plt.show()

evaluate_models(models
```

OUTPUT :-




GRU Model Predictions vs Actual
Actual
GRU Predictions
600
550
500
450
400
350
300
0
5
10
15
20
25


LSTM Model Predictions vs Actual
Actual
LSTM Predictions
600
550
500
450
400
350
300
0
5
10
15
20
25

PRACTICAL : 10

AIM:

Attention-based Image Captioning

In this practical, you will build an encoder-decoder model with an attention mechanism for generating image captions. The focus will be on understanding how to use pre-trained Convolutional Neural Networks (CNNs) as encoders and implementing attention mechanisms in decoders to highlight important image regions. This task will provide insights into the connection between vision and language tasks using deep learning models. You will work with the Flickr8k Image Caption Dataset. Steps to Follow

1. Load the Flickr8k Image Caption Dataset:
   a. Preprocess the dataset.
   b. Split the dataset into training, validation, and test sets.
2. Preprocess Text Data (Caption):
   a. Tokenize the captions and build a vocabulary to map words to indices.
   b. Pad sequences to ensure all captions are of equal length for training.
3. Load and Use Pre-trained CNNs as Encoders:
   a. Use pre-trained models like VGG16, ResNet, or InceptionV3 to extract image features.
   b. Pass the images through the CNN to obtain features that will serve as inputs to the decoder.
4. Define the Encoder:
   a. Use the pre-trained CNN as an encoder to extract features from images.
   b. Modify the CNN's output layer to match the size of the decoder's input.
5. Define the Decoder:
   a. Use an RNN-based decoder (e.g., LSTM or GRU) to generate captions from the image features.

b. Implement attention mechanism to focus on relevant image regions during caption generation.

6. Attention Mechanism:

a. Implement the Bahdanau or Luong attention mechanism:

- Bahdanau: Calculates attention weights as a function of encoder-decoder hidden states.
- Luong: Computes attention weights based on a weighted context vector derived from the encoder's output.

b. Integrate the attention mechanism into the decoder to control the focus on different image regions when generating captions.

7. Training the Model:

a. Train the encoder-decoder model using the Flickr8k dataset.

b. Use cross-entropy loss to optimize the model's performance on caption prediction.

c. Monitor training metrics such as loss and BLEU score on the validation set.

8. Evaluate the Model:

a. Generate captions for images in the test set and compare with ground truth captions.

b. Calculate BLEU scores to evaluate the similarity between generated captions and the ground truth.

This practical will enhance your ability to work with vision-language tasks and deepen your understanding of how attention mechanisms work in neural networks.

CODE :-

```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
from tensorflow.keras.applications import InceptionV3
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.layers import Embedding, LSTM, Dense, AdditiveAttention, Input, Dropout
from tensorflow.keras.models import Model
import os
import pickle

# Load pre-trained InceptionV3 model for feature extraction
base_model = InceptionV3(weights='imagenet', include_top=False)
encoder_model = Model(base_model.input, base_model.output)

def extract_features(image_path):
    img = tf.keras.preprocessing.image.load_img(image_path, target_size=(299, 299))
    img = tf.keras.preprocessing.image.img_to_array(img)
    img = np.expand_dims(img, axis=0)
    img = tf.keras.applications.inception_v3.preprocess_input(img)
    features = encoder_model.predict(img)
    return features

# Sample dataset
captions = ["a cat sitting on the floor", "a dog running in the park"]
tokenizer = Tokenizer()
tokenizer.fit_on_texts(captions)
```

```
vocab_size = len(tokenizer.word_index) + 1

sequences = tokenizer.texts_to_sequences(captions)
padded_sequences = pad_sequences(sequences, padding='post')

# Define decoder with attention
class AttentionDecoder(tf.keras.Model):
    def init(self, vocab_size, embedding_dim, units):
        super(AttentionDecoder, self).init()
        self.units = units
        self.embedding = Embedding(vocab_size, embedding_dim, mask_zero=True)
        self.lstm = LSTM(units, return_sequences=True, return_state=True)
        self.attention = AdditiveAttention()
        self.fc = Dense(vocab_size, activation='softmax')

    def call(self, features, captions):
        x = self.embedding(captions)
        hidden_state, carry_state = self.lstm(x)
        context_vector = self.attention([hidden_state, features])
        output = self.fc(context_vector)
        return output

decoder = AttentionDecoder(vocab_size, embedding_dim=256, units=512)

# Compile and train the model (dummy training for demonstration)
optimizer = tf.keras.optimizers.Adam()
loss_object = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=False)
```

```
def loss_function(real, pred):
    mask = tf.math.logical_not(tf.math.equal(real, 0))
    loss = loss_object(real, pred)
    mask = tf.cast(mask, dtype=loss.dtype)
    loss *= mask
    return tf.reduce_mean(loss)


# Dummy training step
@tf.function
def train_step(features, captions):
    with tf.GradientTape() as tape:
        predictions = decoder(features, captions)
        loss = loss_function(captions[:, 1:], predictions)
    gradients = tape.gradient(loss, decoder.trainable_variables)
    optimizer.apply_gradients(zip(gradients, decoder.trainable_variables))
    return loss


# Function to generate captions
def generate_caption(image_path):
    features = extract_features(image_path)
    input_seq = [tokenizer.word_index['<start>']]
    caption = []
    for _ in range(20):  # Max caption length
        sequence = pad_sequences([input_seq], maxlen=padded_sequences.shape[1], padding='post')
        predictions = decoder(features, sequence)
        predicted_id = np.argmax(predictions[0, -1, :])
        if predicted_id == tokenizer.word_index['<end>']:
            break
```

```
        caption.append(tokenizer.index_word[predicted_id])
        input_seq.append(predicted_id)
    return ' '.join(caption)

print("Model ready for training and caption generation")
```

OUTPUT :-

```
Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/inception_v3/inception_v3_weights_tf_dim_ordering_tf_kernels_notop.h5
87910968/87910968 ━━━━━━━━━━━━━━━━━━━━ 1s 0us/step
Model ready for training and caption generation
```